MASSACHUSETTS INTITUTE OF TECHNOLOGY

VLSI PUBLICATIONS

VLSI Memo No. 89-566
October 1989

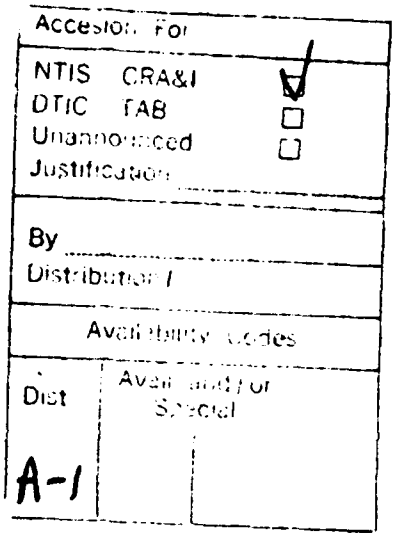
Performance Tradeoffs in Multithreaded Processors

Anant Agarwal

Abstract

High network and memory latencies in large-scale multiprocessors can cause a significant drop in processor utilization. Overlapping computation from alternate processes with memory accesses in multithreaded processors can reduce processor idle time. A multithreaded processor maintains multiple process contexts in hardware and can switch between them in a few (say, zero to 16) cycles. This paper proposes an analytical performance model for multithreaded processors that includes cache interference, network contention, and context-switching overhead effects. The model is validated through our own simulations and by comparison with previously published simulation results. Our results indicate that processors can substantially benefit from multithreading, even in systems with small caches. Large caches yield close to full processor utilization with as few as 2 to 4 processes, while small caches require two to four times more processes. Increased network contention due to multithreading has a negligible effect on performance, and the context switching overhead sets a limit on the best possible utilization.

Microsystems Technology Laboratories
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Room 39-321
Telephone (617) 253-0292



Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825, and by grants from the Sloan Foundation and IBM.

Author Information

Agarwal: Laboratory for Computer Science, Room NE43-418, MIT, Cambridge, MA 02139. (617) 253-1448.

Performance Tradeoffs in Multithreaded Processors

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

High network and memory latencies in large-scale multiprocessors can cause a significant drop in processor utilization. Overlapping computation from alternate processes with memory accesses in multithreaded processors can reduce processor idle time. A multithreaded processor maintains multiple process contexts in hardware and can switch between them in a few (say, zero to 16) cycles. This paper proposes an analytical performance model for multithreaded processors that includes cache interference, network contention, and context-switching overhead effects. The model is validated through our own simulations and by comparison with previously published simulation results. Our results indicate that processors can substantially benefit from multithreading, even in systems with small caches. Large caches yield close to full processor utilization with as few as 2 to 4 processes, while small caches require two to four times more processes. Increased network contention due to multithreading has a negligible effect on performance, and the context switching overhead sets a limit on the best possible utilization.

1 Introduction

As we build larger and larger parallel machines, the proportion of processor time actually spent in useful work keeps diminishing. There are two reasons for the decreasing processor utilization. First, the cost of each memory access increases because network delays increase with system size. As we push VLSI technology and architecture to achieve higher processor clock rates, wire delays will become predominant making communication and memory accesses even more expensive. Second, as we strive for greater speed-ups in applications through fine-grain parallelism, the number of network transactions also increases. Synchronization and process management further reduce the actual time spent doing useful work, but we do not consider these here.

A method of improving processor utilization is to multithread the processor. The idea is to switch the processor to a new thread and perform useful computation while the previous thread's memory request is being satisfied. While multithreading has usually meant cycle-by-cycle interleaving of instructions from different processes, we will apply the same term to non single-cycle interleaved rapid context-switching processors as well. Several previous processor designs used this technique to mask communication latencies or to utilize deep pipelines effectively, e.g. [1, 2, 3, 4, 5, 6, 7]. By multithreading a processor such that an instruction from a different thread can be initiated every cycle (or every few cycles), pipeline bubbles due to pipeline dependencies or processor stalls due to memory latency can be obviated. Processors in message passing multicomputers often maintain multiple processes per node to overlap communication latencies by rapid switching between processes [8, 9].

1.1 The APRIL Processor

We are designing and implementing a multithreaded processor APRIL as part of the ALEWIFE multiprocessor project-at MIT. ALEWIFE is a large-scale, cache coherent machine. Some of the previous multithreaded processor designs, such as Halsteads MASA[3], Monsoon [5], Kaminsky and Davidson work [1], or the Hep [6], in addition to hiding network latencies, used multithreading as a means of utilizing deep pipelines by avoiding single process resource conflicts, and suffered from poor single thread performance. APRIL encourages single process execution in the pipeline using pipeline bypass paths and compiler pipeline optimization to achieve high single thread performance. APRIL has a register file organized into several register frames (or windows) that can store multiple process contexts and obviate register saves and restores across context switches. Each process uses no more than one frame (actually two frames are used per process – the frame adjacent to the current one is for trap handling). A context switch to a process whose context is currently stored in one of the register frames on the processor is effected in a small number of cycles.

The space of processes is virtual, the mapping of process contexts to register frames is maintained and effected in software. In other words processor register frames act a software-controlled cache on the virtual space of process contexts.

The current implementation of APRIL uses the SPARC processor [10] with several modifications. The register set is divided into several frames that are conventionally used as register windows [11] for speeding up procedure calls. SPARC permits the use of these frames for context switching because the bumping of the frame pointer uses a separate instruction and is not necessarily tied to the procedure call. In our design, a process uses just one frame for all its procedures and the registers must be saved and restored across procedure calls, unless inter-procedure register allocation is used.

Fast trap handling, facilitated using the same mechanisms as fast context switches, also allows a quick processor response to external events and we exploit this feature for several purposes. In ALEWIFE, adopting a RISC style approach, the cache controller can generate a processor interrupt for software handling of cases such as coherence directory overflow, special message arrival, or buffer overflow, which significantly simplifies the controller design.

Rapid-trap-handling also allows efficient handling of Futures [12] and full-empty bit [6] traps. Our other modifications to SPARC include redefining the tag bits for Futures support, and alternate space addressing and trapping to support full-empty bits for fine-grain synchronization. Rapid context switching also makes the use of strong coherence protocols feasible, because the processor can switch to a new thread while the acknowledgment to outstanding memory/cache transactions is awaited. The cache/network controller in each ALEWIFE multiprocessor node can activate either the trap line to the processor for lengthy transactions, or the wait line for busy waiting on short transactions such as cache misses to non-remote memory.

1.2 Limitations of Multithreading

There are limits to the improvement achievable through multithreading the processor. Most important, the application must display sufficient parallelism so multiple threads can be assigned to each processor. Provided sufficient parallelism exists, the improved processor utilization

must be traded off against the negative cache and network effects.[1] Multiple simultaneously-active processes interfere with each other in the cache and give rise to a higher cache miss rate and hence a higher network access rate. Similarly, the higher network traffic generated by a fully utilized processor will result in increased network latencies due to network contention. A multithreaded processor design must address the tradeoff between higher utilization through overlapping network access and the increased cache miss rates and network contention.

1.3 Overview

Our analysis is aimed at quantitatively understanding the performance tradeoffs involved in multithreading a processor. What are the limits to the improvement in processor utilization as we increase the number of processes? How do cache and network design impact these limits? To answer these questions, we derived a model of multithreaded processor performance that takes into account the deleterious cache and network effects. This model provides insights into the relationships between the various factors involved, and also performance estimates on expected processor utilization indicating the domains of feasibility of multithreaded processors.

We derive a simple expression for processor utilization that takes into account network contention effects, cache interference effects and context switching overhead. As an indication of the results we obtained, for a set of default parameters given in Table 1, we found that when the number of processes is increased from one to three the processor utilization increases from 0.3 to 0.71, but the corresponding improvement in going from 3 to 5 is just 0.71 to 0.88 due to increased cache interference and context switching overhead.

Clearly, our performance predictions come with a caveat. The model neglects several important concerns, such as the *availability of enough parallel threads*, *register file management*, impact on the clock cycle, and context-switch decision making. Because these parameters are closely related to the characteristics of parallel applications and implementation constraints, the ultimate test of the success of multithreaded processors can only be in an actual system implementation. The APRIL processor architecture and the associated software system design as part of the ALEWIFE multiprocessor project is aimed at investigating these issues in more detail. However, the model presented in this paper can still be used to evaluate tradeoffs in the design process. Furthermore, driving the model with parameters measured from real parallel applications does lend credibility to our results.

The rest of the paper is organized as follows. Section 2 presents the multithreaded processor performance model. This model includes a network model described in Section 3 and a fine-grain multiprogrammed cache model in Section 4. Section A in the appendix extends the model to include several related effects. Validations of the model are in Section 5. Section 6 uses this model to analyze the tradeoffs in multithreaded processors. Section 7 compares our results to previous analyses of multithreaded processors and presents more validations. Section 8 presents directions for future work and the current status of our project.

[1] We will assume for the purpose of this study that the processor includes a cache, where coherence is maintained either by using some form of scalable directory scheme as in ALEWIFE, or purely in software.

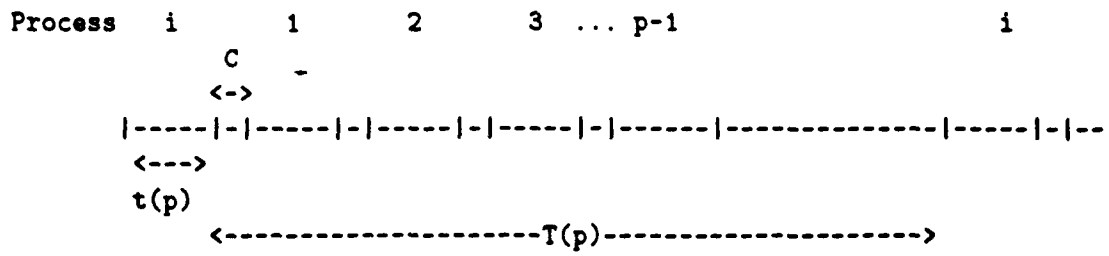


Figure 1: Hiding network latency by multithreading the processor.

2 A Multithreaded Processor Performance Model

A model for a multithreaded processor must represent the tradeoff between increased processor utilization due to overlapping network access with useful execution, and the resulting higher cache miss rate and network latency. For the purpose of this analysis we assume that the processes active on a processor have similar miss rates and that between misses the processes are executing useful instructions. The processor cycles spent in context-switching are considered wasted.

If there are p active processes on a processor (or the number of hardware contexts), let the time between misses for each process be $t(p)$, the time to satisfy a miss be $T(p)$, and the time wasted in context-switching be C. (Exponentially distributed times between misses and network service times can also be assumed, and is discussed further in Section 2.1.) A process executes useful instructions for $t(p)$ cycles, suffers a miss, and then waits $T(p)$ cycles for the miss request to be satisfied before it can proceed. As depicted in Figure 1, some of this network service time can be overlapped with useful processor execution.

With p available processes, and no overhead, effective processor utilization is

$$Util(p) = \frac{p\,t(p)}{t(p) + T(p)}$$

Context switch overhead can be factored in easily. We assume that the miss rate is independent of the context switch overhead, and that the instructions executed during the context switch do not cause any cache misses. This is a reasonable assumption because the code executed during the context switch is either cache resident due to frequent reuse, or is hardwired. The context-switch overhead is independent of the number of hardware contexts in our model. Say, each context switch sufferes an overhead of C processor cycles. The utilization equation remains the same for all p such that

$$p\,[t(p) + C] < t(p) + T(p)$$

because the number of useful cycles during the interval $t(p) + T(p)$ is still $pt(p)$. Otherwise, the utilization becomes limited by the context-switch time. For every $t(p)$ useful cycles, the processor wastes C context-switch related cycles, yielding a limiting utilization of

$$Util(p) = \frac{t(p)}{t(p) + C}, \quad for\ p \geq \frac{t(p) + T(p)}{t(p) + C}$$

If $m(p)$ is the miss ratio, and context switches to new processes are forced on each miss, $t(p)$ is simply the inverse of the miss rate. In practice we might force a context switch only on a miss to a nonlocal memory module. The utilization then becomes,[2]

$$\begin{aligned} Util(p) &= \frac{p}{1+T(p)m(p)}, \quad for\ p < \frac{1+T(p)m(p)}{1+Cm(p)} \\ &= \frac{1}{1+Cm(p)}, \quad for\ p \geq \frac{1+T(p)m(p)}{1+Cm(p)} \end{aligned} \tag{1}$$

We now need to derive expressions for both $m(p)$ and $T(p)$, which are functions of the number of processes. Let us start with summarizing the terms used thus far, and the assumptions used in our analyses.

p	Degree of processor multiprogramming
$T(p)$	Time (or number of processor cycles) to satisfy a network request
$t(p)$	Time between misses
$m(p)$	Cache miss ratio
$Util(p)$	Processor utilization
C	Context-switching overhead in processor cycles

2.1 Assumptions

Our analysis will include the following general assumptions and simplifications.

- We assume all processes sharing the processor have similar properties in terms of their miss rates and working set sizes.
- Network accesses happen only on cache misses. In a multiprocessor cache there are a few other cases where a network access is required, such as invalidations. Their effect can be incorporated into the analysis by adding a constant m_{inv} to the miss rate.
- Initially, we do not consider the effect of processes sharing data in the cache. In a real system, the cache interference will be smaller if some fraction of the working sets of the processes are shared. While low levels of data sharing occur in many applications, sharing of instruction blocks and read-only data is expected (and must be encouraged). Furthermore, affinity scheduling disciplines that favor same-processor execution of threads operating on overlapping data sets will also increase the shared fraction in the cache, and we are investigating several such methods. It then becomes necessary to include the effect of sharing. The miss rate model is extended in Section A.1 by reducing the size of the

[2] If the context-switch overhead is considered part of $t(p)$ and if it contributes to the miss rate, the processor utilization for $pt(p) < t(p) + T(p)$ is given by

$$Util(p) = \frac{p(1-Cm(p))}{1+T(p)m(p)}$$

and for $pt(p) \geq t(p) + T(p)$ is given by

$$Util(p) = \frac{t(p)-C}{t(p)}$$

individual process working sets by the shared fraction, and amortizing the cost of fetching the shared blocks over all the threads.

- In the simple model discussed in this paper, the effect of the multithreaded miss rate due to program nonstationary behavior is not included. A process suffers nonstationary misses when it it renews parts of its working set. When a process returns to the processor, some fraction of its working set is renewed. If this effect is not accounted for, the analysis will incorrectly count some nonstationary misses as multithreading related misses. We have seen that this effect is not significant for most applications. Nevertheless, Section A.2 modifies the model to account for this effect.

- Our processor utilization model assumes fixed time intervals between context switches and fixed network service time. We could also assume a fixed probability of a miss on any cycle of processor execution, leading to geometrically distributed time intervals between context switches (or exponential for the continuous case) with mean $t(p)$, and exponentially distributed network service times with mean $T(p)$. Due to the loop nature of programs exponential time between misses is hard to justify, and in lightly loaded networks the network service time will be fairly uniform. Nevertheless, the processor utilization can be derived from a simple M/M/1//M queueing model [13] for a finite population of size p, a single queueing server (the processor) with exponential service time with mean $t(p)$, and the network modeled as a delay center with exponential service times with mean $T(p)$, as one minus the probability the queueing server (the processor) is idle, or

$$Util(p) = 1 - \frac{1}{\sum_{q=0}^{q=p} (\frac{t(p)}{T(p)})^q \frac{q!}{(q-k)!}}$$

Some other assumptions that have been traditionally made in network and cache analysis will also be made and we shall point these out in the relevant sections. Although not essential to our model, to simplify some of our network latency expressions, we will assume that the multithreaded processor operates in the range where $T(p) >> p$. That is, sufficient contexts are not available to allow a context switch every cycle and completely mask network latency. This region of values is reasonable for several reasons. First, it is hard to imagine a large number of available hardware contexts on the processor between which very low-overhead context switches can occur. Put another way, we are assuming that the caches are useful enough that the miss rate is reasonably small, and that a large number of contexts is unnecessary. Similarly, for the network model, we will assume that the interesting operating range is where $T(p) >> t(p)$, or else the incentive to multithread the processor does not exist. A typical sampling of parameters might be: $T = 100$, $1/m(p) = t(p) = 20$, and $p = 4$.

3 Network Latency

This section derives the network latency as a function of the number of active threads in the processor. We will use a packet-switched multistage interconnection network with $k \times k$ switches ($k = 2$ in all our analyses). Infinite buffering at the switch nodes is assumed. Simulation experiments [14] have shown that as few as 4 packet buffers at each switch node can approach infinite buffer performance. The network model makes the usual assumptions of uniform traffic rates from all the nodes, and uniformly distributed and independent destinations. Packet size is B times the network channel width, the number of network stages is n, and the memory

access time is M cycles. We will drop the dependence of t and T on the number of processes for notational convenience.

We will start with the interconnection network model proposed by Kruskal and Snir [14]. The probability of a request at a network port is p/T, when $T >> t$. If M is the memory access time, and assuming B sized packets, the overall network access time is given by,

$$T = \left[1 + \frac{bin(pB/T)(1 - 1/k)}{2(1 - pB/T)}\right] 2n + M + B - 1$$

The delay is basically the memory delay plus twice the delay through the network of n stages. The delay at each switch stage is one plus the queueing delay. This model has been used extensively in the literature. We also ran simulations with address traces of parallel applications and found that the predictions of the model were generally accurate.

With 2x2 switches, T becomes

$$T = \left[1 + \frac{B}{4(T/Bp - 1)}\right] 2n + M + B - 1$$

Solving for T, we get,

$$T = \frac{Bp}{2} + n + \frac{M}{2} + \frac{1}{2}\sqrt{B^2p^2 + M^2 + 4n^2 + 4Mn - 4Bpn - 2BpM + 2B^2pn}$$

The above expression for T can be greatly simplified if we assume $T >> Bp$, and $T >> M$. The above assumptions simply state that both memory access time and the message traversal time through one network switch stage are small compared to the delay through the network. Note that these assumptions stem out of our motivation for multithreading processors, which is the high latency of network accesses in a large-scale multiprocessor system. Because the unloaded network latency $2n$ constitutes a large part of T, we can also use $2n >> p$. When a significant part of the network latency is caused by contention delays, or when a large number of hardware contexts are available for rapid-context switching, we cannot make this assumption, and the complete expression must be used for T. We first write T as,

$$T = \frac{Bp}{2} + n + \frac{M}{2} + n\sqrt{\frac{B^2p^2}{4n^2} + \frac{M^2}{4n^2} + 1 + \frac{M}{n} - \frac{Bp}{n} - \frac{BpM}{2n^2} + \frac{B^2p}{2n}}$$

Neglecting second order terms in Bp/n and in m/n we get,

$$T \approx \frac{Bp}{2} + n + \frac{M}{2} + n\sqrt{1 + \frac{M}{n} - \frac{Bp}{n} + \frac{B^2p}{2n}}$$

Taking the first-order terms in the binomial expansion of the square-root expression, we get a simplified expression for the effective network time with p active processes.

$$T \approx 2n + M + \frac{B^2p}{4} \tag{2}$$

The above expression for T implies that when the unloaded network access time is much greater than the number of processes, and if the blocksize is small, then the effective network

access time with p processes is not significantly different. The reason this is true is that when the load becomes higher due to a large p, so does the network latency, yielding a lower effective network access rate due to the negative feedback effect. However, when the network load becomes larger, i.e., when the $B^2p/4$ term becomes significant in comparison to $2n + M$, the network contention delay becomes important.

4 A Multithreaded Cache Model

We now derive the effective cache miss rate when there are p processes sharing the cache. The cache model must account for the increase in cache miss rate as the number of processes increases. We use the following notation from [15].

- S Direct-mapped cache size in terms of the number of cache sets (or rows)
- B Network message length or block size
- u Working set size of each process in blocks
- τ The size of the time quantum used in measuring the working set
- c The collision rate used in computing intrinsic interference
- v The size of the set of blocks a process leaves behind in the cache when it is switched out, or the size of the *carry-over set*

We will start with a model for multiprogrammed caches. Such a model was derived by Stone and Thiebaut [16] for two processes, and a similar model for an arbitrary number of processes was derived by Agarwal, Horowitz, and Hennessy [15]. The model in [15] included the following assumptions: Mapping of addresses to cache sets is random. Processes display the working set model of program behavior in that in any given interval of time, a small set of blocks is in active use. The time interval τ used in measuring the working set is assumed to be large enough for the working set of a process to be brought into the cache. The multiprogramming model further assumed that the context-switch interval is greater than τ. This assumption simplified the analysis greatly and was required so that the process could fetch its entire working set into the cache during its time slice on the processor.

Our analysis here cannot avail of the last assumption, which is generally valid in single processor environments, where the system maintains long context switch times when possible to avoid cache thrashing. However, by its very nature, a multithreaded processor switches contexts over very short intervals of time. In fact, our analysis assumes that switches are forced on every miss. Clearly, one miss can hardly replenish the entire working set of a process in the presence of several intervening processes. Therefore, only a fraction of a process's working set can then be retained in the cache in the steady state.

As an intuitive example, let us suppose context switches happen every cycle. Let the number of processes active in the cache be large enough that a returning process i finds *none* of its blocks in the cache. As defined previously, the carry-over set of a process i is the set of blocks of process i left in the cache when it is switched out. On the miss, the process i fetches exactly one block into the cache, yielding a steady-state carry-over set size of one irrespective of its actual working set size.

The rest of this section estimates the multiprogrammed cache miss rate for very short context switching intervals. For comparison, Appendix B presents a simplified miss rate model for direct-mapped caches with context switch intervals large enough to allow a process to replenish its working set completely. We first review relevant portions of the analytical cache model found

in [15].

4.1 Review

The *steady-state* cache miss rate is the sum of three components: the non-stationary, the intrinsic interference, and the extrinsic interference components. The component of the miss rate caused by multiprocessor sharing invalidations, m_{inv}, can also be included. It is not explicitly modeled in our analysis here, but its effect can be incorporated into the fixed non-stationary miss rate component. The non-stationary component is the cost of bringing in blocks into the cache due to first-time references. Our analyses here will simply lump this miss-rate component into the term denoted m_{ns}. The intrinsic interference component represents the misses caused when the blocks from the working set of a given processes interfere with each other in the cache. The extrinsic interference component represents the misses caused due to multiprogramming related interference. In this paper, we will refer to the non-stationary and multiprocessor invalidation misses as the *fixed* miss rate components.

For a direct-mapped cache, with the uninterrupted execution interval large enough to bring in the entire working set of the process, the equation for the intrinsic miss rate component presented in [15] for a direct-mapped cache ($d = 1$) is:

$$\begin{aligned} m_{intr} &= \frac{c}{\tau}\left[u - Sbin(u, \frac{1}{S}, d = 1)\right] \\ &= \frac{c}{\tau}u\left[1 - \left(1 - \frac{1}{S}\right)^u\right] \end{aligned} \tag{3}$$

where c, the collision rate, is independent of cache size (for cache size greater than the working set size) and is treated as a constant in our analysis. The term $bin(u, \frac{1}{S}, d)$ is the binomial distribution[3] and represents the probability that d blocks from the working set of size u map into one of the S cache sets. u minus S times the value of this distribution at $d = 1$ yields the number of blocks of the process that collide with each other in the cache. This number times the collision rate divided by the interval yields the miss rate.

The fine-grain context-switching model requires the computation of the carry-over set size v of a process. As defined previously, the carry-over set of a process i is the set of blocks of the process left in the cache when it is switched out. The size of this set for process i is v when it is the only process active in the cache. Because of replenishment, v is independent of p.

If blocks are randomly mapped to cache sets, then the carry-over set can be derived from the working set size u as

$$v = S\left[1 - (1 - \frac{1}{S})^u\right] \tag{4}$$

where the probability that a given block does not map into a given set is $(1 - 1/S)$, which when raised to power u is the probability no block from the process's working set maps into that set. One minus the above quantity is the probability there is at least one block mapped to a cache set, which when multiplied by S yields the number of cache sets used.[4]

[3] $bin(u, \frac{1}{S}, d) = \binom{u}{d}\left(\frac{1}{S}\right)^d\left(1 - \frac{1}{S}\right)^{u-d}$

[4] For simplicity, our definition for the carry over set here does not make the correction suggested in [15] for the blocks left behind in the cache by a process that are more likely to be purged due to intrinsic interference. The difference between the model here and the more accurate one is very small.

The analysis in the next section focuses on modified models for the interference components when the context switch interval becomes small.

4.2 Miss Rate with Small Context-Switch Intervals

When the context-switching interval becomes very small, the size of the carry-over set also reduces. Let $v'(p)$ be the *steady-state* carry-over set size with p processes sharing the cache, when the context-switch time is small. Multiprogramming the cache with a small interval increases the intrinsic interference component of misses. The process effectively sees a smaller cache and hence its intrinsic interference component increases to $m'_{intr}(p)$. Having to replenish part of the effective carry over set each time the process is scheduled to run on the processor adds in the fine-grain-context-switching component of misses denoted $m'_{cs}(p)$. ($m_{cs}(p)$ is used in Appendix B to represent the coarse-grain-context-switching miss rate of the cache).

4.2.1 Computing $m'_{cs}(p)$

We will first derive the context-switching component of the miss rate $m'_{cs}(p)$. Let $v'_j(p)$ be the carry-over set size before steady state is reached for the j^{th} occurrence of process i on the processor, and let $t(p)$ be the number of references before a context switch for each process. The probability of a first-time reference of a block during each access is u/τ, which is computed as the ratio of the number of unique references u to time quantum tau. Then, the total number of unique blocks accessed by the intervening processes is

$$(p-1)t\frac{u}{\tau}$$

If the references of the intervening $p-1$ processes are randomly distributed throughout the cache, then the probability that a given block of the intervening processes maps on top of a block of process i is $v'_j(p)/S$. From this we can approximate the number of block s of process i displaced from the cache as

$$\frac{v'_j(p)}{S}(p-1)t\frac{u}{\tau}$$

The above expression assumes that the probability of purging a block of process i stays constant throughout the intervening $(p-1)$ processes. An accurate expression for the purged blocks is $v[1-(1-1/S)^{(p-1)tu/\tau}]$ if t is large enough that a sizable fraction of the carry-over set is purged. In practice, however, the simpler expression gives almost identical results to the more complicated expression.

Similarly, when the process i is resumed, it replenishes its set of blocks in the cache. If it runs for t cycles, accessing tu/τ unique blocks during this time, the number of previously-displaced blocks fetched into the cache is

$$\frac{v-v'_j(p)}{v}t\frac{u}{\tau}$$

where $(v-v'_j(p))/v$ is the probability that a referenced block is not already present in the cache. As before, a more accurate expression for the number of new blocks can be derived as $(v-v'_j(p))[1-(1-1/v)^{tu/\tau}]$ when t is large.

In the steady state, the number of purged blocks must equal the number of replenished blocks, and $v'_j(p) = v'(p)$. Thus,

$$\frac{v'_j(p)}{S}(p-1)t\frac{u}{\tau} = \frac{v - v'_j(p)}{v}t\frac{u}{\tau}$$

Simplifying, the steady-state carry-over set size with p processes is,

$$v'(p) = \frac{v}{1 + v\frac{(p-1)}{S}} \tag{5}$$

The above equation indicates that for context-switching intervals that are small compared to τ, the carry-over set size is independent of t. For large t, when the probability of purging or replenishing a block in the carry-over set is no longer constant over the time interval between occurrences of a process, the carry-over set-size does indeed depend on t, and the more complex expressions shown earlier, or their higher order approximations, must be used.

We can make several useful observations from the above estimate of the carry-over set size $v'(p)$ of each process in a multithreaded processor. When the cache is very large ($S >> u$), we find that $v'(p) \approx v$, and $v \approx u$, indicating that the cache can comfortably hold the entire working set of every process. When the $S = v$, effective cached working set of each process is v/p. That is, each process gets only a fraction of its working set inversely proportional to the number of processes. Finally, when $S << v$, the cached working set is limited to the cache size divided by the number of processes, or $S/(p-1)$.

The corresponding miss rate due to context switching $m'_{cs}(p)$ is the rate at which blocks in the carry-over set of a process are purged by intervening processes.

$$m'_{cs}(p) = v'(p)\frac{(p-1)}{S}\frac{u}{\tau} \tag{6}$$

4.2.2 Computing $m'_{intr}(p)$

Now, let us compute the increase in the intrinsic interference component. In Equation 3 we will recompute the the number of colliding blocks. The new number of colliding blocks is simply the difference between the working set size u and the number of blocks in the carry-over set $v'(p)$ that do not collide with other blocks of the process i. Given random placement of blocks in the cache, the number of non-colliding blocks in the multiprogrammed cache will decrease by the same fraction as the carry-over set, that is, by the fraction $v'(p)/v$. We can thus estimate the number of blocks that do not collide with other blocks of process i as

$$u\left(1 - \frac{1}{S}\right)^u \frac{v'(p)}{v}$$

and substituting in Equation 3 the intrinsic interference component of the miss rate with p processes becomes,

$$m'_{intr}(p) = \frac{c}{\tau}\left[u - u\left(1 - \frac{1}{S}\right)^u \frac{v'(p)}{v}\right] \tag{7}$$

We now derive the increase in the intrinsic interference component due to multithreading as,

$$m'_{intr}(p) - m_{intr} = \frac{c}{\tau} u \left(1 - \frac{1}{S}\right)^u \left(1 - \frac{v'(p)}{v}\right) \tag{8}$$

Intuitively, in the above equation, $u(1 - 1/S)^u$ is the number of blocks of a processes that do not collide in a single-process cache, $(1 - v'(p)/v)$ is the fraction of these blocks that become involved in collisions in a multithreaded cache, and c/τ is the average number of times a colliding cache block causes cache misses.

Thus, the net increase in the miss rate due to multiprogramming with small context-switch intervals, denoted $m'(p)$, is the sum of $m'_{cs}(p)$ and $m'_{intr}(p) - m_{intr}$, or,

$$\begin{aligned} m'(p) &= m'_{cs}(p) + m'_{intr}(p) - m_{intr} \\ &= v'(p)\frac{(p-1)\,u}{S}\frac{}{\tau} + \frac{c}{\tau} u \left(1 - \frac{1}{S}\right)^u \left(1 - \frac{v'(p)}{v}\right) \end{aligned} \tag{9}$$

where $v'(p)$ and v are obtained from Equations 5 and 4 respectively. The overall miss rate can be represented as the sum of the multiprocessor invalidation component, nonstationary component (both constants for this analysis), single process interference component (Equation 3), and the component due to multithreading (Equation 9).

$$m(p) = m_{inv} + m_{ns} + m_{intr} + m'(p) \tag{10}$$

4.3 Model Summary and Simplifications

The expressions for network latency derived in Equation 2 and that for cache miss rate from 10 can be substituted into Equation 1 for the processor utilization.

The cache model can be significantly simplified to obtain the nature of the dependence of the miss rate on the number of processes. We will first obtain a simplified expression for the ratio of $m'(p)$ and m_{intr}. Replacing $v'(p)$ and v by their respective formulae in terms of u, S, and p, and making the approximation

$$\left(1 - \frac{1}{S}\right)^u \approx \left(1 - \frac{u}{S}\right)$$

the ratio of the multithreading induced miss rate to the intrinsic miss rate to:

$$\frac{m'(p)}{m_{intr}} \approx \frac{(p-1)\left(1 + \frac{1}{c}\right)}{1 + (p-1)\frac{u}{S}} \tag{11}$$

Furthermore, in the region where $(p-1)u << S$, we can further simplify the ratio to:

$$\frac{m'(p)}{m_{intr}} \approx (p-1)\left(1 + \frac{1}{c}\right) \tag{12}$$

The overall miss rate is then,

$$m(p) \approx m_{inv} + m_{ns} + m_{intr} + m_{intr}(p-1)\left(1 + \frac{1}{c}\right) \tag{13}$$

where, as before,

$$m_{intr} = \frac{c}{\tau}\left[u - u\left(1 - \frac{1}{S}\right)^{u}\right]$$

or simplified for small $\bar{u}/S$,

$$m_{intr} = \frac{c}{\tau}\frac{u^2}{S} \tag{14}$$

5 Validation of the Model

We conducted some experiments to verify that our approximations in the cache model were indeed valid. The validations use three traces: IVEX0 is a sample trace of a DEC program, Interconnect Verify, checking net lists in a VLSI chip (under VMS). LocusRoute is a global router for VLSI standard cells. The SIMPLE code models hydrodynamic and thermal behavior of fluids in two dimensions.

We extracted a trace of one processor from the multiprocessor traces and replicated it multiple times to simulate the effect of a multithreaded processor trace. IVEX0 is itself a single processor trace, SIMPLE is a 64 processor trace and LocusRoute has 8 processors. The process identifier of each thread was hashed into the address used to index into the cache to avoid systematic collisions with the same addresses of the other processes. While, using a multithreaded trace generated from individual traces of all the processors would have been a more realistic situation, validation is hard because of statistical variations between the individual traces and the difficulty of deriving real parameterized traces with varying numbers of processes.

Figure 2 shows the model predictions and simulation results. The plots show the increase in miss rate due to multithreading for caches with $S = 16K$, and block size 16 bytes.[5] In general the predictions were quite good. LocusRoute and SIMPLE compared well with simulations, while the IVEX0 trace yielded a poorer match. In both LocusRoute and SIMPLE the miss rate increase with number of processors is small because of the smaller working set size compared to IVEX0. Part of the reason IVEX0 has a higher working set size is that the IVEX0 trace, unlike the other two traces, includes operating system references. Our conjecture why the IVEX trace yielded a poorer match was because of an optimistic assumption regarding the non-stationary component. When the nonstationary correction was not specifically included in the model, the analytical model results tended to be much higher than the simulation. The same was true for SIMPLE. The nonstationary correction assumes that the nonstationary blocks fetched in each time quantum do not contribute to multithreading interference misses, which is an optimistic assumption.

We found that the approximate miss rate model using Equation 12 is valid up to only about 6 processes, which is as expected because the approximation is invalid when $(p-1)u/S$ is no longer negligible compared to 1. The more accurate from Equation 11 is virtually indistinguishable from the most accurate model of Equation 10 (shown in the Figure).

[5] For the purpose of validation, the correction to the model for non-stationarity in the program is included in the graphs. The SIMPLE graphs use a cache with $S = 64K$ because of systematic collisions in smaller caches.

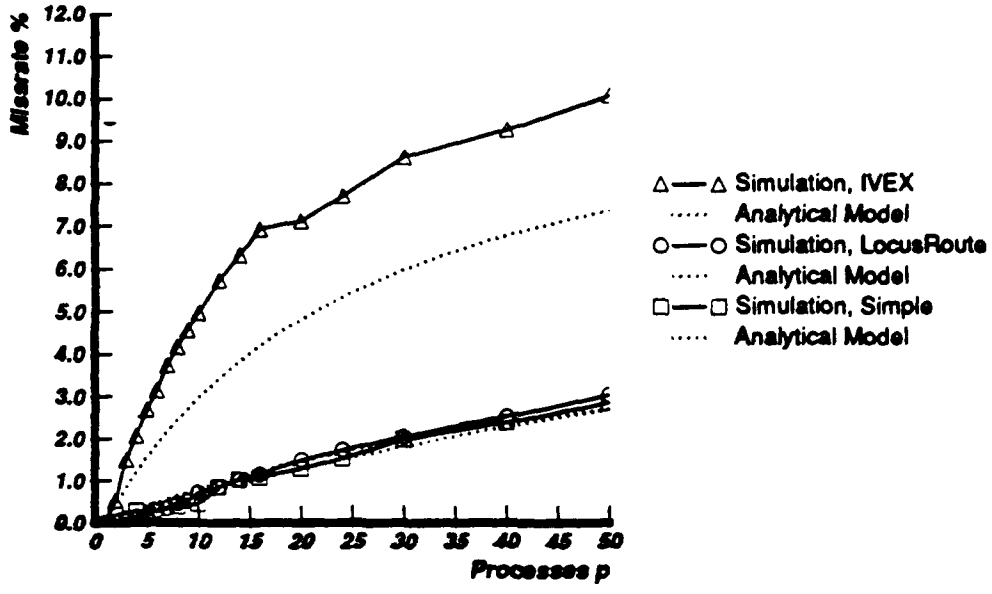


Figure 2: Increase in the miss rate of a fine-grain multiprogrammed cache.

6 Performance Implications of Multithreading

Multithreading impacts performance in three ways. A higher degree of multithreading typically suffers a higher network latency and a higher cache miss rate. The larger number of processes allows some fraction of the network delay to be overlapped with computation often increasing the processor utilization, to the extent that the overhead of context switching does not detract from the benefits. Let us first make some general observations from the model.

The increase in network latency is proportional to the number of processes and to the square of the block size. If the unloaded base latency is large the contention effects can be ignored.

From the simple model, to first order, the processor utilization does not change due to multithreading if the intrinsic interference component of the cache miss rate is high, which can happen if the cache is small compared to the working set size of a process. This is easily seen from the processor utilization Equation 1 with $m(p)$ substituted from Equation 13. A high m_{intr} component implies that the processor utilization is roughly,

$$Util(p) \approx \frac{p}{T(p)m_{intr}(p-1)\left(1+\frac{1}{c}\right)} \quad for \;\; p > 1$$

However, if the ratio of processes to cache size $(p-1)/S$ is held constant, the utilization increases linearly with the number of processes, despite a high interference component. That is, to support more processes with the same high interference miss rate, the cache should be made proportionally larger, as can be seen by substituting $m_{intr} = cu^2/S\tau$ in the above equation,

$$Util(p) \approx \frac{p}{T(p)\frac{cu^2}{S\tau}(p-1)\left(1+\frac{1}{c}\right)} \quad for \;\; p > 1$$

On the other hand, if the interference miss rate component is small compared to the sum of the fixed miss rate components m_{ns} and m_{inv}, utilization will increase linearly with the number

of processes given the same cache size. The utilization equation becomes

$$Util(p) \approx \frac{p}{1 + T(p)(m_{ns} + m_{inv})} \quad for \ p > 1$$

Our experience with multiprocessor simulations with parallel applications has been that the fixed miss rate component is generally large relative to the intrinsic interference component. Both higher multiprocessor cache invalidation rates and smaller working sets of fine-grain processes contribute to this observation, and future multiprocessors are increasingly likely to operate in this range.

For large caches, the cache model also supports the intuition that the interference component of the miss rate is proportional to the number of processes and inversely proportional to the cache size.

The intrinsic and the extrinsic components of the miss rate are related to the square of the working set sizes of the processes. Working set changes will dramatically impact cache performance. Hence effort put into compacting data words efficiently into cache blocks to increase the fraction of used words in cache lines will be effort well spent. Process sharing of blocks on a processor must be encouraged for the same reason. Reduction in the private working sets of the processes yields proportionally greater benefits in cache miss rate and processor utilization due to the square law dependence on the working set size.

As investigated in Section A.1, threads can benefit substantially by sharing data in the cache. Sharing helps in two ways: the working set sizes that contribute to multithreading interference misses are effectively reduced, and the cost of fetching in shared blocks are amortized over all the threads.

Context-switching overhead becomes important only when the number of processes is large enough to completely overlap the network latency with processor execution (including the processor cycles spent in context-switching). The reason is that for a smaller number of processes, the processor cycles wasted in context switching would otherwise have been spent in waiting on the network.

We will now use the accurate model to estimate the effects of varying the parameters on processor utilization using the default parameters given in Table 1. The miss rate and working set size defaults are typical of the applications we have measured. Figure 3 shows the degree to which each component impacts overall processor utilization. For the parameters chosen, network contention has little effect. Cache miss rate is the predominant factor, and the context switching overhead becomes important only when the number of processes is large enough to allow complete masking of network latency.

6.1 Effect of Network Contention

Let us first investigate the impact of the increased network contention on processor utilization keeping the other factors such as cache miss rate at a constant value and the overhead at zero. Figure 4 shows the effect of contention for three different values of the base network and memory latency $2n + M$.

For small latencies, as in the case for a small multiprocessor system, one to 4 processes are sufficient to yield close to perfect utilization when the cache effects and overhead are ignored. As network latencies become larger, say in a large-scale machine, more processes become useful. The effect of increased network contention caused by more processes is small for the packet size

Parameter	Value
Context switching overhead	4 cycles
Network and memory latency	100 cycles
Fixed miss rate	2%
Cache block size	16 bytes
Average message size	4 words
Process working set size	250 blocks
Cache size ($S = 4K$)	64Kbytes

Table 1: Default analysis parameters.

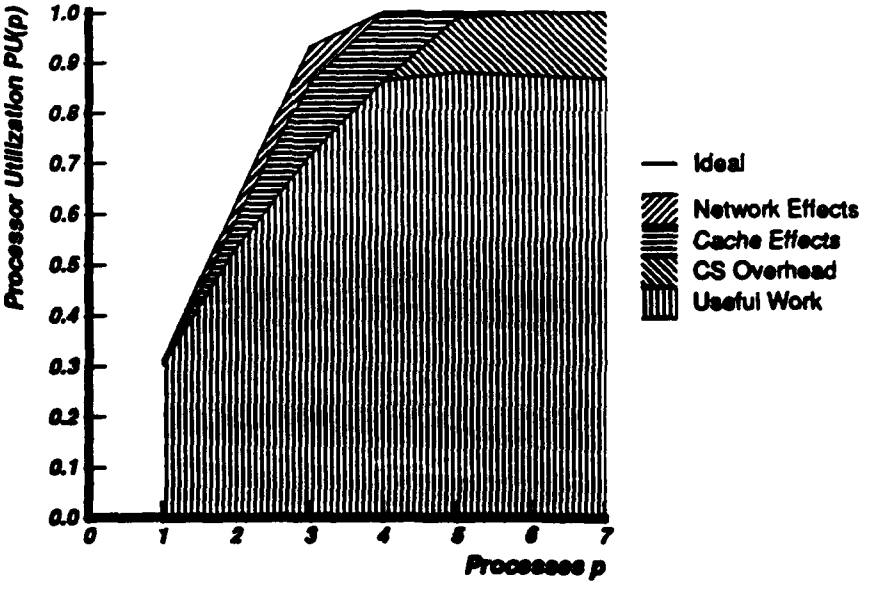


Figure 3: Relative sizes of the various components that affect processor utilization.

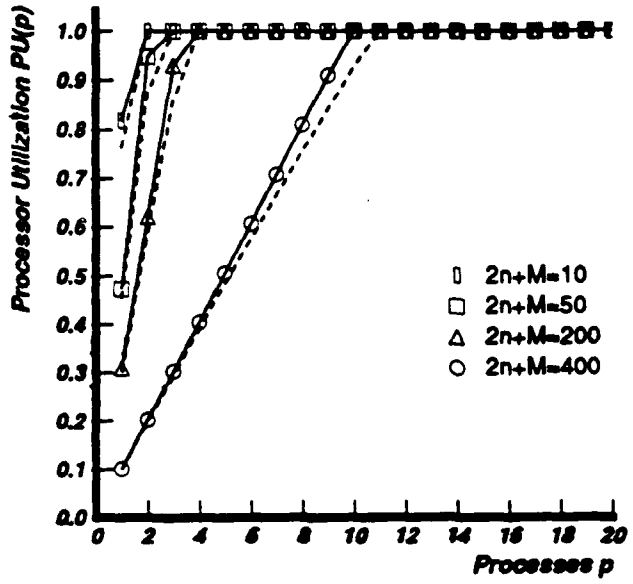


Figure 4: Impact of increased network contention on processor utilization. The solid curves represent the ideal utilization without the extra network contention and the dashed curves correspond to the lowered utilization due to network contention.

chosen. Because the contention is related to the square of the packet size, greater packet sizes will cause the network effects to become more important.

Our model used a packet-switched multistage interconnection network, whose clock cycle was assumed to be independent of the number of processors. In practice, switching delays in driving longer wires in large systems will cause the network latencies to increase. Similarly, mesh or cube connected direct networks suffer longer latencies due to the increased number of hops. When network latencies increase multithreading the processor becomes even more useful.

6.2 Effect of Context-Switching Overhead

Figure 5 shows the effect of context switching overhead for the default set of parameters. Context switching overhead gives rise to a limiting utilization of $1/(1 + m(p)C)$. Once utilization reaches its maximum value for $p = (1 + m(p)T(p))/(1 + m(p)C)$, increasing it further decreases the utilization because the miss rate becomes worse. We see this effect when the overhead is greater than one in Figure 5.

The effect of overhead will have a significant impact on processor design. Obtaining very small overheads will mean significant hardware additions to the processor. On a switch, the program counters and the processor status word must be saved, the register frame pointer must be bumped, the pipeline must be restarted. Program counters could be saved in a stack much like per-processor register frames. The effect of pipeline flushing is harder to minimize because register instructions from the previous process cannot write to the current frame and bypass registers might be storing some hidden state. Additional hardware can mitigate some of these problems, but then the impact on cycle time must also be considered. If slightly higher overheads can be tolerated, the task of the processor designer will be significantly simplified.

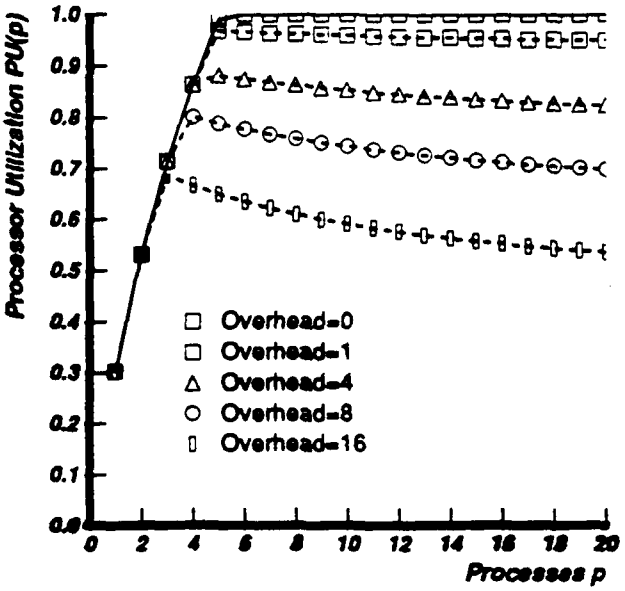


Figure 5: Impact of context-switching overhead on processor utilization.

Our results do indicate that higher overheads can be tolerated if the number of processes needed to fully utilize the processor is small. For example, with a 4 cycle overhead and 8 processes the utilization is still close to 90%. Put another way, because the product of the miss rate and the overhead determine utilization, a relatively high context-switching overhead can be tolerated if the miss rate is correspondingly low.

6.3 Cache Effects

Figure 6 shows the effect of the fixed cache miss rate for the default parameter set. As few as two or three processes can fully mask network latency for fixed miss rates of up to 1%. A higher nonstationary miss rate requires many more processes to obtain comparable utilization, although the utilization bears an almost linear relationship to the number of processes because of the low interference component. Recall that the multithreading interference overhead is determined by the product of the intrinsic interference component and the number of processes.

When the working set size is larger (Figure 7) the rate of increase of processor utilization with the number of processes is much smaller. The reason is that the larger working set suffers a higher interference component, which causes the overall miss rate to increase in proportion to the number of processes. Figure 7 clearly shows the dramatic impact of working set size on processor utilization.

Figure 8 assesses the impact of cache size. Cache size effects are small for caches greater than 64Kbytes ($S = 4096$) because large caches can comfortable sustain the working sets of multiple processes (see discussion following Equation 5). In large caches the interference component is much smaller than the nonstationary component, making multithreading always useful. Smaller caches suffer more interference. When the cache is small, the relative effect of increasing the number of processes is smaller because higher interference detracts from the benefit of having more processes. For example, just three processes achieved 70% processor utilization in a 64Kbyte cache ($S = 4096$), while a comparable utilization required 6 processes in a 16 Kbyte

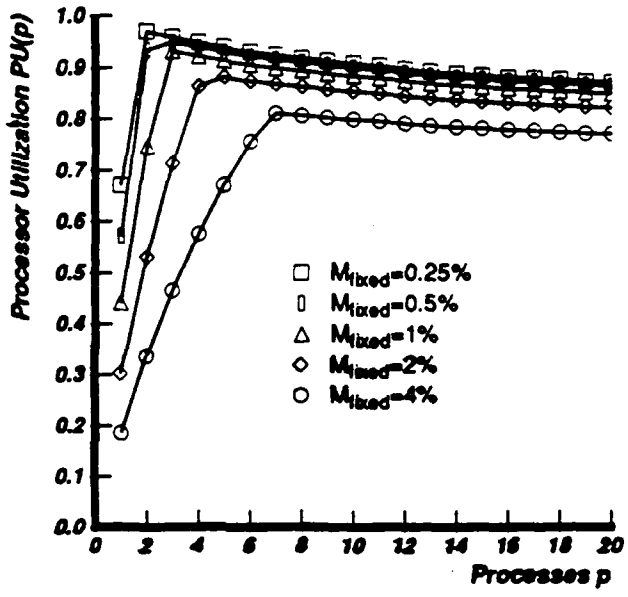


Figure 6: Impact of increased cache miss interference on processor utilization. Varying fixed miss rate.

cache ($S = 1024$). The marginal benefit of adding more processes keeps decreasing as caches become smaller.

6.4 Effect of Network and Memory Latency

How large a network and memory latency can we mask using multithreading? Suppose we had 8 processes available to run on the processor. Then as the flat region of the corresponding curve in Figure 9 depicts, multithreading can mask all network latencies up to a maximum of 200 cycles. The maximum utilization is determined by the context switch overhead and is roughly 0.85 for 8 processes. If the number of available processes is only 2, then a latency increase from 25 to 100 cycles causes the utilization to drop from 0.9 to 0.55.

7 Related Work

Bert Halstead advocated the use of multithreaded processors as the processing nodes in multiprocessors and presented some analyses to estimate the benefits of multithreading [17], but without accounting for network contention, context switching overhead, fixed cache miss rate, and sharing in the cache. The cache miss rate as a function of processes is modeled as the cache size per process raised to some power (both powers -1 and -0.5 are tried), and the cache size per process is chosen to be the total cache size divided by the number of processes. Our analyses shows that a similar relationship between the cache miss rate, cache size, and number of processes is valid only when the fixed miss rate is negligible compared to the interference component, and when cache sizes are smaller than the individual process working sets (see discussion following Equation 5). However, in practice, the fixed miss rate for large caches usually dominates over the interference components, and process working sets are rarely bigger than cache size.

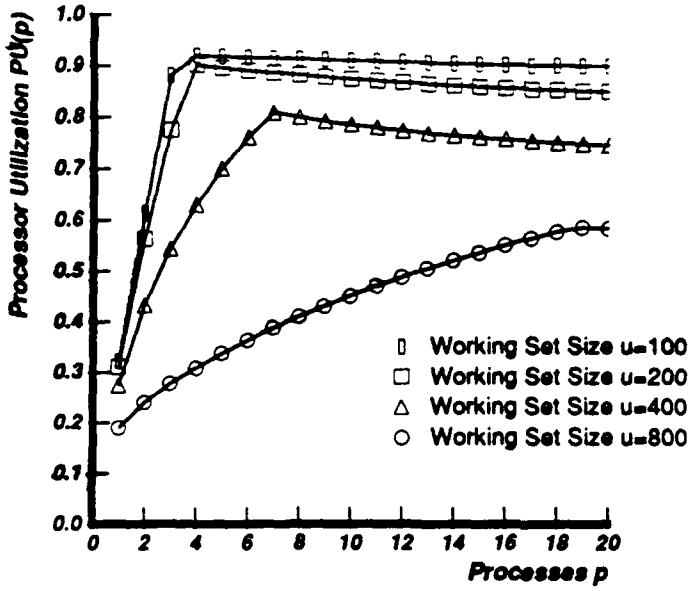


Figure 7: Impact of increased cache miss interference on processor utilization. Varying the working set size.

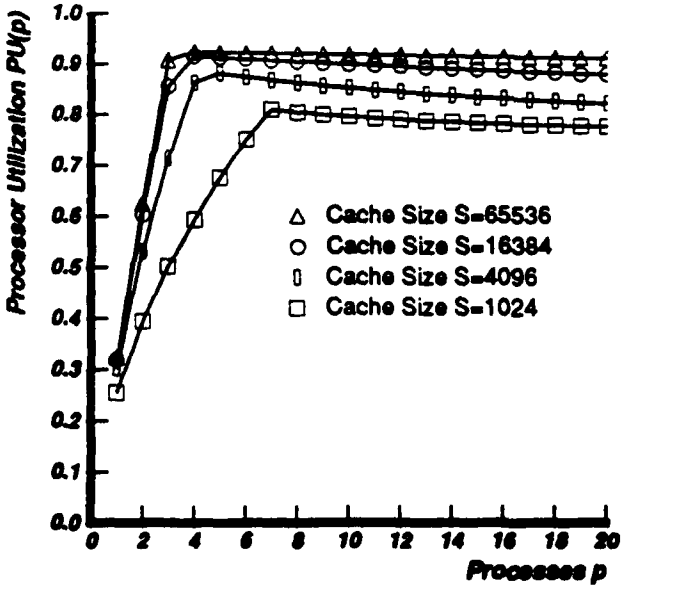


Figure 8: Impact of increased cache miss interference on processor utilization. Varying cache size.

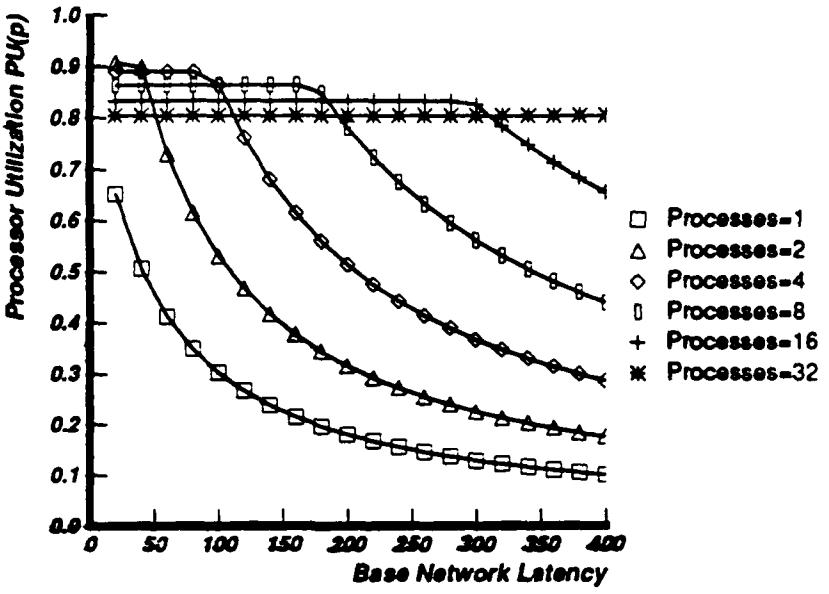


Figure 9: Impact of network and memory latency on processor utilization while considering miss rate increase due to multithreading. Varying $2n + M$.

Halstead uses a queueing model to obtain the probability that at least one process is available to run on the processor, which is simply the processor utilization. Although we assumed constant switch times and constant service times dependent on the number of processes for our results, we showed how exponential distributions for the above intervals can be used in our model in Section 2.1. In Halstead's analyses more contexts always yielded better processor utilization, while in ours, more contexts often harmed utilization because of smaller time intervals with more contexts given a fixed context switching overhead.

Weber and Gupta conducted a simulation study to explore the benefits of multithreading processors using add. ess traces from parallel applications. In the parameter ranges of the traces they measured, our analytical results yield the same conclusions. When the network latency is fairly small (say less than 50 cycles), very few (2 to 4) contexts are sufficient to yield close to complete processor utilization. Often, when the active contexts were increased beyond a certain number the utilization dropped due to the effects of switching overhead.

We conducted a validation experiment with the LocusRoute trace that was also used in Weber and Gupta's simulation study. We measured parameters from the trace, such as process working sets and the constant c, and derived processor utilizations for parameter ranges considered in the simulation study. We found a good match between the model predictions and the simulation numbers. For instance consider Table 2. Analytically computed processor utilizations for varying context-switch intervals, network latencies, and processes are shown, along with the corresponding numbers from the simulation study in parentheses. The same numbers are plotted in Figure 10; the simulation graphs from [18] are on the left. We observe that the model was fairly accurate in predicting not only the relative effects of varying several parameters, but also the absolute processor utilizations.

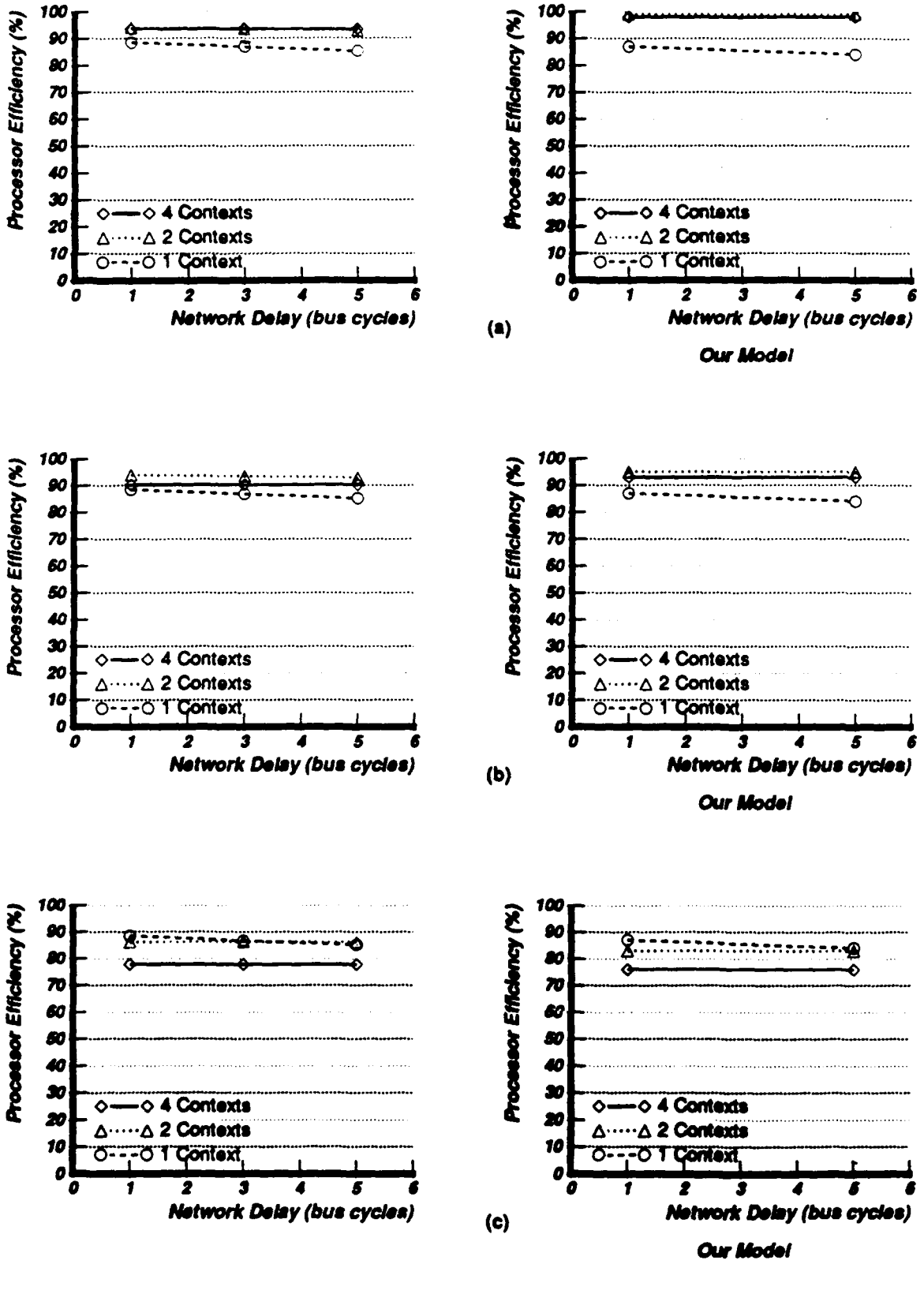


Figure 10: Comparing the model with simulations. (a) Switching overhead 1, (b) 4, and (c) 16.

p	Overhead 1				4				16			
	Latency 18		23		18		23		18		23	
1	0.87	(.885)	0.84	(.851)	0.87	(.885)	0.84	(.851)	0.87	(.885)	0.84	(.851)
2	0.99	(.940)	0.99	(.928)	0.95	(.940)	0.95	(.928)	0.83	(.861)	0.83	(.862)
4	0.98	(.936)	0.98	(.933)	0.93	(.904)	0.93	(.901)	0.76	(.778)	0.76	(.778)

Table 2: Processor utilizations predicted by the model and measured from simulations for context-switch overheads 1, 4, and 16, memory latencies 18, and 23, and processes 1, 2, and 4.

8 Directions for Future Work and Status

The model can be extended in several ways. Direct network models can be used in place of our multistage network. Multithreading will have a greater role to play in such networks due to potentially higher network latencies. The impact of the number of hardware contexts on the clock cycle of the processor can be modeled. An interesting area for research is the impact of grain size and degree of parallelism on working set sizes of threads. A reduction in working set size of threads is expected with increased parallelism, a trend our simulations confirm. Such a decrease implies that (1) increasing the multithreading degree of the processor might not proportionally increase the miss rate, (2) working set sizes of fine-grain threads are expected to be relatively small, encouraging the design of single-level, integrated, smaller, faster caches for shared data.

What is a good scheduling strategy for threads? More threads on few processors implies good utilization of these processors, potential sharing of data in the cache, and communication locality, but suffers from cache interference and poor load balance. A performance study of these effects would be useful. When is it best to switch register frames? When should we swap contexts from register frames? In our current SPARC-based design switching between frames does not come for free; Swapping process contexts from the processor is much more expensive.

The design of a multithreaded processor must solve several new problems in cache coherence protocol design unique to multithreading. For example, one problem the ALEWIFE multiprocessor simulator exposed is the pinging of cache blocks between two threads on the same processor.[6] A thread switched out on a cache miss might find that the block was purged by an intervening thread soon after the block was fetched from memory. This cycle can repeat indefinitely if the two processes access blocks that map into the same cache location. A similar thrashing can occur between threads in different processors. The directory-based cache coherence protocol in ALEWIFE has been modified to ensure forward progress in such situations.

Another potential problem can arise if busy-wait spinning is the mode of synchronization used. By not relinquishing the processor, a thread busy waiting on a lock can preclude the release of the lock held by another thread awaiting its turn on the processor. In APRIL, synchronization is achieved using full-empty bits that trap the processor on an access to a locked block. Other busy-waiting mechanisms we are exploring include adaptive backoff [19] with blocking.

The current status of the design is that the ALEWIFE simulator, ASIM, composed of the APRIL processor, the cache, and network, is operational. The Mul-T [20] compiler and programming system and has been adapted for APRIL. Experimentation with several parallel ap-

[6] Initially we conjectured that such pinging would occur rarely, but simulations showed we were overly optimistic.

plications including the Viterbi search in speech recognition, an object-oriented logic simulation system, particle in cell code, Boyer, are in progress. Initial simulations with parallel applications show that the fixed miss rate is fairly high causing multithreading to be potentially useful. Several related issues such as suitable sizes of each frame, number of frames, and hardware versus software frame management, rapid context switching in the presence of coprocessors, and adaptive backoff spinning versus blocking are the focus of ongoing research.

9 Conclusions

Multithreading a processor to allow overlapping memory and network access times is a useful technique to improve processor utilization. We have developed a novel analytical model for multithreaded processors in the presence of caches. The model also accounts for network contention and context-switching overhead.

Our results indicate that multithreading is useful under a wide range of cache, processor and network parameter variations. Increased network contention effects due to multithreading are small, while cache contention can significantly hurt the performance. Our analyses show that lowering the working set sizes of the individual threads dramatically improves processor performance. Large caches yield close to full processor utilization with as few as 2 to 4 processes, while small caches force the use of two to four times as many processes for the same high utilization. If portions of the working sets of the threads are shared, the negative cache effects of multithreading are mitigated, for the cost of fetching in shared blocks is amortized over several threads. Multithreading is shown to be beneficial when the fixed cache miss rate is large compared to the intrinsic interference component of misses. Multiprocessor simulations of parallel applications indicate that such behavior is increasingly likely as we exploit parallelism at finer grains.

10 Acknowledgments

The research reported here has benefited significantly from discussions with Bert Halstead, Tom Knight, Greg Papadopoulos, Juan Loaiza, Bill Dally, Steve Ward, and Randy Osborne. My gratitude to the ALEWIFE team, including David Kranz, David Chaiken, Kirk Johnson, Kiyoshi Kurihara, John Kubiatowicz, Beng Hong Lim, Gino Maa, and Dan Nussbaum, for their contributions and efforts. Wolf-Dietrich Weber and Anoop Gupta supplied their simulation data for the LocusRoute trace. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, and by grants from the Sloan foundation and IBM.

A Extensions to the Model

We will now suggest the changes that need to be made to the cache model to account for sharing between processes and non-stationary behavior. We will start with Equation 9 for the component of misses due to multithreading,

$$m'(p) = v'(p)\frac{(p-1)}{S}\frac{u}{\tau} + \frac{c}{\tau}u\left(1-\frac{1}{S}\right)^{u}\left(1-\frac{v'(p)}{v}\right)$$

where,

$$v'(p) = \frac{v}{1 + v\frac{(p-1)}{S}}$$

$$v = S\left[1 - \left(1 - \frac{1}{S}\right)^{u}\right].$$

A.1 Process Sharing

Portions of the working sets of processes that are shared are not impacted by interference from other processes. Furthermore, the nonstationary component of the miss rate m_{ns} is reduced because just one process need fetch the shared blocks into the cache for the first time. Let u_{priv} be the number of blocks that are private to the given process out of the total u blocks in its working set.

The multithreading miss rate becomes,

$$m'(p) = v'_{priv}(p)\frac{(p-1)}{S}\frac{u_{priv}}{\tau} + \frac{c}{\tau}u\left(1 - \frac{1}{S}\right)^{u}\left(1 - \frac{v'_{priv}(p)}{v_{priv}}\right)\left(\frac{u_{priv}}{u}\right) - m_{ns}\frac{(u-u_p)}{u}\frac{(p-1)}{p} \tag{15}$$

where,

$$v'_{priv}(p) = \frac{v_{priv}}{1 + v_{priv}\frac{(p-1)}{S}}$$

$$v_{priv} = S\left[1 - \left(1 - \frac{1}{S}\right)^{u_{priv}}\right].$$

The first term in Equation 15 is the context-switching component m'_{cs} as before. Because the shared cache blocks are not impacted by context switching, all the working set related variables are replaced by the corresponding private working set and derived variables.

The derivation of the second term, however, is not as straightforward. Recall, in Equation 8, $u(1-1/S)^u$ is the number of blocks of a process that do not collide in a single-process cache, and $(1-v'(p)/v)$ is the extra fraction of blocks that become involved in collisions in the multithreaded cache. We now multiply this fraction by (u_{priv}/u) so that only the private blocks in the working set are included in this extra component.

The third term arises because the nonstationary component of the miss rate decreases. In this term, $m_{ns}(u-u_p)/u$ is an estimate of the proportion of the nonstationary miss rate due to shared references for a single process, which when divided by p yields the actual nonstationary component of the miss rate for blocks shared between p processes. The term $m_{ns}\frac{(u-u_p)}{u}\frac{(p-1)}{p}$ must be subtracted from the single process nonstationary miss rate component m_{ns} to yield the corresponding component for p processes with sharing.

Because the effective working set is reduced to $u - u_p$, sharing in the cache can significantly decrease the multithreading interference. It is also conceivable that the nonstationary reduction term is greater than the increase in miss rate due to multithreading, and actually result in improved cache hit rates. A similar anomalous miss rate reduction was observed in a multiprogrammed system due to process sharing of operating system structures [21].

A.2 Non-Stationary Effects

As Appendix B shows, modeling the effect of non-stationarity in the addressing behavior of a program for multiprogrammed caches, where the time quanta are large enough to bring in the entire process working set, is straightforward. However, the effect on a finely multithreaded cache are harder to estimate.

Let u_{ns} be the number of blocks that are renewed by a process every τ. The the probability a block of a process is dead on a context switch is $(u - u_{ns})/u$. The effect of nonstationarity on the extrinsic component of the miss rate is to reduce the effective number of blocks left behind by a process in the cache when it is switched out. The rationale is that the dead blocks do not give rise to extrinsic interference. Then,

$$m'_{cs}(p) = v'_s(p)\frac{(p-1)}{S}\frac{u}{\tau}$$

where

$$v'_s(p) = v'(p)\frac{u - u_{ns}}{u}$$

The impact on the intrinsic interference component can be estimated as follows: The first miss of a colliding block that is actually dead due to nonstationarity in the program must not be counted as a intrinsic interference miss. The number of such misses is the product of the number of colliding blocks and the probability a colliding block is dead. The probability of a dead colliding block is estimated as u_{ns}/u. Therefore, the intrinsic interference miss rate due to multithreading

$$m'_{intr} - m_{intr} = \frac{cu}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right) - \frac{u}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right)\frac{u_{ns}}{u}$$

or

$$m'_{intr} - m_{intr} = \frac{u}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right)\left[c - \frac{u_{ns}}{u}\right]$$

B A Multiprogrammed Cache Model

A model for multiprogrammed caches where the context-switch interval is large enough to allow a process to completely replenish its working set before being switched out, was derived in [15]. This model predicts the cache miss as a function of the number of processes active, the degree of associativity of the cache, the cache size, and the working set sizes. For the important special case of direct-mapped caches the model can be greatly simplified. We will modify the model to account for nonstationary behavior, or for those program blocks that are renewed across context switches. From [15], the number of multiprogramming misses that process i suffers on being rescheduled in a cache with p active processes, degree of associativity D, number of sets S, is

$$S\sum_{d=0}^{d=D} bin(u_i, \frac{1}{S}, d)\sum_{e=0}^{e=u_{i'}} MIN(d, e+d-D)bin(u_{i'}, \frac{1}{S}, e)$$

In the above expression, the term $bin(u_i, \frac{1}{S}, d)$ is the binomial distribution and represents the probability that d blocks from the working set of process i of size u_i map into one of the S

cache sets. $u_{i'}$ represents the sum of the working-set sizes of the all the other processes, and $bin(u_{i'}, \frac{1}{S}, e)$ represents the corresponding probability that e blocks of the other $(p-1)$ processes map into any cache set. The number of misses divided by the time quantum of execution of the process (say, τ) yields the context-switching miss rate $m_{cs}(p)$.

If the working set size of each process is u, substitute $u_i = u$, and $u_{i'} = (p-1)u$. Furthermore, if the cache is direct-mapped ($D = 1$), the above equation simplifies to

$$m_{cs}(p) = \frac{S}{\tau} bin(u, \frac{1}{S}, 1) \left[1 - bin((p-1)u, \frac{1}{S}, 0)\right]$$

Recalling, $bin(u, \frac{1}{S}, d) = \binom{u}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d}$, we get,

$$m_{cs}(p) = \frac{S}{\tau} \frac{u}{S} \left(1 - \frac{1}{S}\right)^{u-1} \left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right] \tag{16}$$

In the above equation, $\frac{u}{S}\left(1 - \frac{1}{S}\right)^{u-1}$ is the probability a set has *exactly one* block of process i mapped onto it in a direct-mapped cache,[7] and $\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right]$ is the probability that at least one of the blocks of the intervening $(p-1)$ processes maps into that cache set purging the block of process i. The product of the above two probabilities and S yields the number of blocks of process i purged. For $S >> pu$ the miss rate component further simplifies to

$$m_{cs}(p) = (p-1)\frac{u^2}{S\tau}$$

The above relation shows that the multiprogramming miss rate in large caches, for small p, is linearly related to the number of processes and to the square of their working set sizes.

We will now contrast this miss rate with the trend in fine-grain multithreading. From Equations 12 and 14 the multithreading induced miss rate

$$m'(p) = \frac{c}{\tau}\left(1 + \frac{1}{c}\right)\frac{u^2}{S}(p-1)$$

Interestingly, the miss rate for fine or coarse grain multiprogramming depends on the square of the working set u and linearly on the number of processes; the difference lies in their constants. That is,

$$\frac{m'(p)}{m_{cs}} = c\left(1 + \frac{1}{c}\right)$$

where typical measured values for c are between 1.5 and 2.5.

[7] As explained in [15], we do not count purges of blocks from sets that have more than one block of process i mapped onto it because such blocks are more likely to be purged by the intrinsic interference within the process itself. A simpler model, albeit inaccurate, might consider all the blocks left behind in the cache by process i, yielding, $\left(1 - \left(1 - \frac{1}{S}\right)^u\right)$ for the probability that a set has *at least one* block mapped to it. Note that both expressions (using exactly one and at least one) simplify to $\frac{u}{S}$ when $u << S$, which is often the case. Simulation with address traces confirmed that the two approximations gave virtually indistinguishable results for large caches.

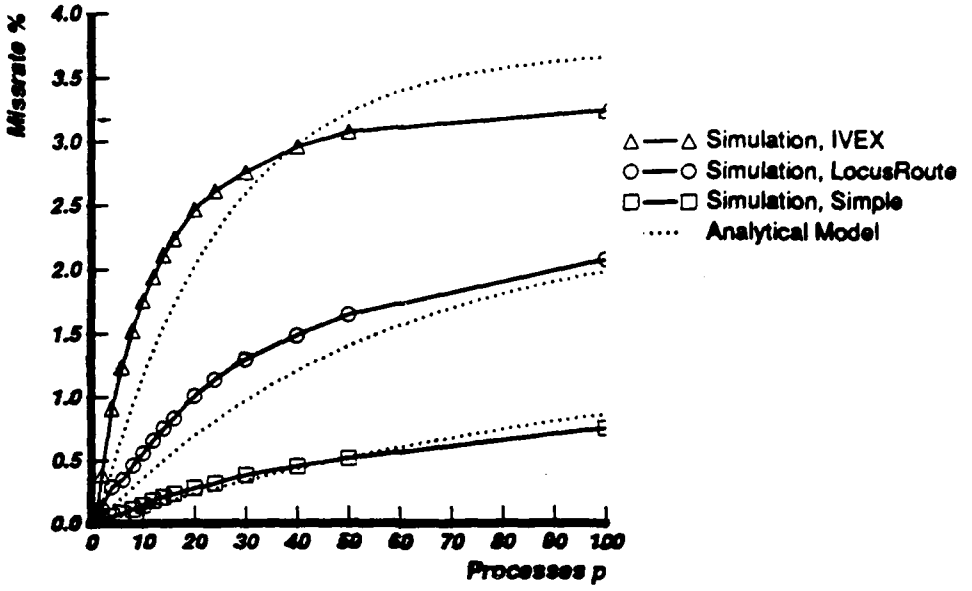


Figure 11: Miss rate of a multiprogrammed cache.

B.1 Correcting the Multiprogramming Model for Nonstationarity

An inaccuracy with the above model was assumption that *every* block left behind in the cache when a process relinquished the processor would be reused by the process on its return. Consequently, for application programs where the working set changed significantly between context switches the model overestimated the multiprogramming misses. This inaccuracy is easily fixed by reducing the working set size of the process i that relinquishes the processor by the fraction of blocks that are renewed by the processes. The number of blocks that are fetched in each time quantum for the first time can be estimated as the total number of unique blocks in a trace divided by the number of time quanta [15]. Let this number be denoted as u_{ns}. Then the miss rate can be written as

$$m_{cs}(p) = \frac{S}{\tau}\left(\frac{u - u_{ns}}{S}\right)\left(1 - \frac{1}{S}\right)^{u-u_{ns}-1}\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right]$$

Figure 11 compares the multiprogramming induced miss rate for the model and simulations with the three traces and cache parameters used in Section 5. The simulations were conducted as follows. We extracted all the references of a single process from a multiprogrammed trace and round-robin scheduled p instances of this single-process trace to simulate a multiprogramming level of p, assuming a time quantum of 10,000 references. Each instance of the trace was assigned a random number as a process identifier (PID). The PID of each constituent process was hashed in with the addresses to randomize the locations in the cache occupied by corresponding references of the p subprocesses. It is easy to see that the model compares well with simulations.

References

[1] W. J. Kaminsky and E. S. Davidson. Developing a Multiple-Instruction-Stream Single-Chip Processor. *Computer*, 66–78, December 1979.

[2] E. S. Davidson. A Multiple Stream Microprocessor Prototype System: AMP-1. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 9–16, IEEE, New York, May 1980.

[3] R.H. Jr. Halstead and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[4] R.A. Iannu ı. *A Dataflow/von Neumann Hybrid Architecture.* Technical Report TR-418, MIT, 545 Technology Square, Cambridge, MA, May 1988.

[5] G.M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* Technical Report TR-432, MIT, 545 Technology Square, Cambridge, MA, August 1988.

[6] B.J. Smith. Architecture and applications of the hep multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[7] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von neumann computing? *ACM*, mar, 1989.

[8] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9–24, August 1988.

[9] W. J. Dally et. al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, IEEE, New York, June 1987.

[10] Robert B. Garner. Sun builds an open risc architecture sparc scalable processsor architecture. *Sun Technology*, 42–55, Summer 1988.

[11] M. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Thesis, Computer Science Division (EECS) UCB/CSD 83/141, University of California at Berkeley, October 1983.

[12] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[13] Leonard Kleinrock. *Queueing Systems.* John Wiley & Sons, 1975.

[14] Clyde P. Kruskal and Marc Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

[15] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, May 1989.

[16] Harold S. Stone and Dominique Thiebaut. Footprints in the Cache. In *Proceedings of ACM SIGMETRICS 1986*, pages 4–8, May 1986.

[17] Jr. Robert H. Halstead. Processor Architecture for Multiprocessors. 1985. Unpublished Report.

[18] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[19] Anant Agarwal and Mathews Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[20] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[21] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.